

CSED211 REPORT

20220124 김문겸

Lab3: Bomb_Lab

<phase_1>

<phase 1>

```
<phase_1>
Dump of assembler code for function phase_1:
0x000000000400ef0 <+0>:      sub    $0x8,%rsp
0x000000000400ef4 <+4>:      mov    $0x402570,%esi → %ESI가 0x402570 대입
0x000000000400ef9 <+9>:      callq 0x4013be <strings_not_equal>
0x000000000400efe <+14>:     test   %eax,%eax → EAX 값 AND 연산
0x000000000400f00 <+16>:     je     0x400f07 <phase_1+23> → je: OF가 0이면 실행됨. <+23> line 23 jump
0x000000000400f02 <+18>:     callq 0x401624 <explode_bomb> → je가 실행X라면 jump하지 못하고 explode_bomb 실행
0x000000000400f07 <+23>:     add    $0x8,%rsp → je 조건문에서 실행됨.
0x000000000400f0b <+27>:     retq

End of assembler dump.
```

<strings_not_equal>

```
Dump of assembler code for function strings_not_equal:
0x0000000004013be <+0>:      push   %r12
0x0000000004013c0 <+2>:      push   %rbp
0x0000000004013c1 <+3>:      push   %rbx
0x0000000004013c2 <+4>:      mov    %rdi,%rbx → %RAX이 %rdi 값 → %rdi, %rsi가 함수 argument
0x0000000004013c5 <+7>:      mov    %rsi,%rbp → %rbp이 %rsi 값
0x0000000004013c8 <+10>:     callq 0x4013a1 <string_length> → %ESI에 값이 들어있을 경우
0x0000000004013cd <+15>:     mov    %eax,%r12d
0x0000000004013d0 <+18>:     mov    %rbp,%rdi
0x0000000004013d3 <+21>:     callq 0x4013a1 <string_length>
0x0000000004013d8 <+26>:     mov    $0x1,%edx
0x0000000004013dd <+31>:     cmp    %eax,%r12d
0x0000000004013e0 <+34>:     jne    0x401420 <strings_not_equal+98>
0x0000000004013e2 <+36>:     movzbl (%rbx),%eax → RAX 값은 %EAX로 저장
0x0000000004013e5 <+39>:     test   %al,%al → %al 값이 0 인지
0x0000000004013e7 <+41>:     je     0x40140d <strings_not_equal+79> → %edx = 0 이 jump
0x0000000004013e9 <+43>:     cmp    0x0(%rbp),%al
0x0000000004013ec <+46>:     je     0x4013f7 <strings_not_equal+57> (if test 결과 0일 경우)
0x0000000004013ee <+48>:     xchg   %ax,%ax
0x0000000004013f0 <+50>:     jmp    0x401414 <strings_not_equal+86> → %edx = 1 이 jump
0x0000000004013f2 <+52>:     cmp    0x0(%rbp),%al
0x0000000004013f5 <+55>:     jne    0x40141b <strings_not_equal+93>
0x0000000004013f7 <+57>:     add    $0x1,%rbx
0x0000000004013fb <+61>:     add    $0x1,%rbp
0x0000000004013ff <+65>:     movzbl (%rbx),%eax
0x000000000401402 <+68>:     test   %al,%al
0x000000000401404 <+70>:     jne    0x4013f2 <strings_not_equal+52>
0x000000000401406 <+72>:     mov    $0x0,%edx edx에 0 대입
0x00000000040140b <+77>:     jmp    0x401420 <strings_not_equal+98>
0x00000000040140d <+79>:     mov    $0x0,%edx edx에 0 대입
0x000000000401412 <+84>:     jmp    0x401420 <strings_not_equal+98>
0x000000000401414 <+86>:     mov    $0x1,%edx edx에 1 대입
0x000000000401419 <+91>:     jmp    0x401420 <strings_not_equal+98>
0x00000000040141b <+93>:     mov    $0x1,%edx edx에 1 대입
0x000000000401420 <+98>:     mov    %edx,%eax → %edx를 %eax에 대입
0x000000000401422 <+100>:    pop     %rbx
0x000000000401423 <+101>:    pop     %rbp
0x000000000401424 <+102>:    pop     %r12
0x000000000401426 <+104>:    retq

End of assembler dump.
```

OF가 0이 되려면,
test에 의해 계산된
(eax)&(eax) = 0 이거나 한
→ 즉, %EAX = 0 이거나 한가?

→ 값이 0x402570의 주소에 들어있는
string을 확인해보자?

→ 확인결과

(gdb) >/s 0x402570
→ I am just a renegade hockey mom.

<string_length>

```
Dump of assembler code for function string_length:
0x0000000004013a1 <+0>:      cmpb    $0x0,(%rdi) 0x0과 rdi의 비교
0x0000000004013a4 <+3>:      je     0x4013b8 <string_length+23> → (%rdi)가 0이면
0x0000000004013a6 <+5>:      mov     %rdi,%rdx
0x0000000004013a9 <+8>:      add     $0x1,%rdx
0x0000000004013ad <+12>:     mov     %edx,%eax
0x0000000004013af <+14>:     sub     %edi,%eax
0x0000000004013b1 <+16>:     cmpb    $0x0,(%rdx)
0x0000000004013b4 <+19>:     jne     0x4013a9 <string_length+8>
0x0000000004013b6 <+21>:     repz    retq
0x0000000004013b8 <+23>:     mov     $0x0,%eax
0x0000000004013bd <+28>:     retq

End of assembler dump.
```

%eax에 0 대입

<+4>에서 %esi 에 0x402570 에 0x402570 에 있는 값을 저장하는 것을 확인할 수 있다. 그리고 %esi 는 <strings_not_equal>의 인자로 들어간다. <strings_not_equal>함수를 역어셈블해서 살펴보면, 이름에서도 알 수 있듯이 string 이 일치하는지 불일치하는지 검사하는 함수이다. 따라서 우리는 %esi 에 들어가는 0x402570 이 가리키는 값을 입력하면 된다. 'x/s'를 이용해서 확인해보면 "I am just a renegade hockey mom."인 것을 알 수 있다.

Answer: "I am just a renegade hockey mom."

(뒤에도 있습니다)

<phase_2>

<phase 2>

```
Dump of assembler code for function phase_2:
0x0000000000400f0c <+0>:  push    %rbp
0x0000000000400f0d <+1>:  push    %rbx
0x0000000000400f0e <+2>:  sub     $0x28,%rsp
0x0000000000400f12 <+6>:  mov     %rsp,%rsi
0x0000000000400f15 <+9>:  callq   0x40165a <read_six_numbers>
0x0000000000400f1a <+14>:  cmpl    $0x0, (%rsp) → rsp ≠ 0 이면 explode_bomb 실행 (1번째 숫자=0)
0x0000000000400f1e <+18>:  jne     0x400f27 <phase_2+27>
0x0000000000400f20 <+20>:  cmpl    $0x1, 0x4(%rsp) → (rsp+4) ≠ 1 이면 explode_bomb 실행 (+4는 첫 번째 숫자, 다음 숫자=1)
0x0000000000400f25 <+25>:  je      0x400f48 <phase_2+60>
0x0000000000400f27 <+27>:  callq   0x401624 <explode_bomb>
0x0000000000400f2c <+32>:  jmp     0x400f48 <phase_2+60>
0x0000000000400f2e <+34>:  mov     -0x8(%rbx), %eax ← rbx에 (rsp+8) 주소 저장
0x0000000000400f31 <+37>:  add     -0x4(%rbx), %eax ← rbp에 (rsp+24) 주소 저장
0x0000000000400f34 <+40>:  cmp     %eax, (%rbx)
0x0000000000400f36 <+42>:  je      0x400f3d <phase_2+49>
0x0000000000400f38 <+44>:  callq   0x401624 <explode_bomb>
0x0000000000400f3d <+49>:  add     $0x4,%rbx → rbx = rsp+8
0x0000000000400f41 <+53>:  cmp     %rbp, %rbx ← rbp = rba, 즉 1번째 숫자 검증
0x0000000000400f44 <+56>:  jne     0x400f2e <phase_2+34>
0x0000000000400f46 <+58>:  jmp     0x400f54 <phase_2+72>
0x0000000000400f48 <+60>:  lea     0x8(%rsp), %rbx
0x0000000000400f4d <+65>:  lea     0x18(%rsp), %rbp
0x0000000000400f52 <+70>:  jmp     0x400f2e <phase_2+34>
0x0000000000400f54 <+72>:  add     $0x28,%rsp
0x0000000000400f58 <+76>:  pop     %rbx
0x0000000000400f59 <+77>:  pop     %rbp
0x0000000000400f5a <+78>:  retq

End of assembler dump.
```

QUESTION
<+34> line에서 ADDet,
<+44> line에서 ADD는
이 기능이 다른가?
→ <+34>는 %eax에 ADD, <+44>는 %rbx = %rsp+8 이 ADD
%rsp는 다른 레지스터라 다르지,
스택 포인터의 역할을 수행하는데
ADD %rsp, %rsp 포인터를 값만큼
증가시킨다.

```
Dump of assembler code for function read_six_numbers:
0x000000000040165a <+0>:  sub     $0x18,%rsp
0x000000000040165e <+4>:  mov     %rsi,%rdx
0x0000000000401661 <+7>:  lea     0x4(%rsi), %rcx
0x0000000000401665 <+11>:  lea     0x14(%rsi), %rax
0x0000000000401669 <+15>:  mov     %rax, 0x8(%rsp)
0x000000000040166e <+20>:  lea     0x10(%rsi), %rax
0x0000000000401672 <+24>:  mov     %rax, (%rsp)
0x0000000000401676 <+28>:  lea     0xc(%rsi), %r9
0x000000000040167a <+32>:  lea     0x8(%rsi), %r8
0x000000000040167e <+36>:  mov     $0x402861, %esi
0x0000000000401683 <+41>:  mov     $0x0, %eax
0x0000000000401688 <+46>:  callq   0x400c30 <__isoc99_sscanf@plt>
0x000000000040168d <+51>:  cmp     $0x5, %eax
0x0000000000401690 <+54>:  jg      0x401697 <read_six_numbers+61>
0x0000000000401692 <+56>:  callq   0x401624 <explode_bomb>
0x0000000000401697 <+61>:  add     $0x18,%rsp
0x000000000040169b <+65>:  retq

End of assembler dump.
```

보여줘서,

0 1 1 2 3 5

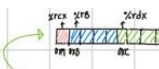
<+14>줄에서, %rsp 와 0 을 cmp 한다. 여기서 (%rsp) = 0 이 아니면 폭탄이 터진다.
<+20>줄에서, %rsp+4 와 1 을 cmp 한다. 여기서 (%rsp+4) = 1 이 아니면 폭탄이 터진다.
여기서 %rsp 는 첫 번째 숫자, (%rsp+4)는 두 번째 숫자를 의미한다. 여기까지 통과를 했다면
rbx 에 rsp+8, 즉 세 번째 숫자와 rbp 에 rsp+24, 즉 네 번째 숫자의 주소를 저장한다. 이후

<+34> line 으로 이동한다. 여기서 eax 에 rbx-8, 즉 rsp 값을 저장한다. 그리고 eax 가 rbx, 즉 세 번째 숫자와 같지 않다면 폭탄이 터진다. 따라서 세 번째 숫자는 첫 번째 숫자와 두 번째 숫자를 더한 값이 되어야 한다. 즉, 피보나치 수열이다! 그리고 여기에 rbx-4, 즉 rsp + 4 값을 더한다. 이후 rbx 에 +4 를 더한다. 즉, 다음 번째 숫자로 넘어가는 것을 의미한다. 그리고 rbp = rbx, 즉 7 번째가 되기 전까지 다시 <+34> 줄로 돌아간다. 즉 2 번의 답은 6 개의 피보나치 수열을 적는 것이 된다.

Answer: 0 1 1 2 3 5

<phase_3>

<phase 3>



```

Dump of assembler code for function phase_3:
0x0000000000400f5b <+0>: sub    $0x18,%rsp
0x0000000000400f5f <+4>: lea    0x8(%rsp),%r8
0x0000000000400f64 <+9>: lea    0x7(%rsp),%rcx
0x0000000000400f69 <+14>: lea    0xc(%rsp),%rdx
0x0000000000400f6e <+19>: mov    $0x4025be,%esi
0x0000000000400f73 <+24>: mov    $0x0,%eax
0x0000000000400f78 <+29>: callq  0x400c30 <_isoc99_sscanf@plt>
0x0000000000400f7d <+34>: cmp    $0x2,%eax
0x0000000000400f82 <+39>: jg     0x400f87 <phase_3+44>
0x0000000000400f87 <+44>: callq  0x401624 <explode_bomb>
0x0000000000400f8c <+49>: ja     0x40108e <phase_3+307>
0x0000000000400f92 <+55>: mov    0xc(%rsp),%eax
0x0000000000400f96 <+59>: jmpq   *0x4025d0(,%rax,8)
0x0000000000400f9d <+66>: mov    $0x69,%eax
0x0000000000400fa2 <+71>: cmpl   $0x22c,0x8(%rsp)
0x0000000000400faa <+79>: je     0x401098 <phase_3+317>
0x0000000000400fb0 <+85>: callq  0x401624 <explode_bomb>
0x0000000000400fb5 <+90>: mov    $0x69,%eax
0x0000000000400fba <+95>: jmpq   0x401098 <phase_3+317>
0x0000000000400fbf <+100>: mov    $0x72,%eax
0x0000000000400fc4 <+105>: cmpl   $0x1d6,0x8(%rsp)
0x0000000000400fcc <+113>: je     0x401098 <phase_3+317>
0x0000000000400fd2 <+119>: callq  0x401624 <explode_bomb>
0x0000000000400fd7 <+124>: mov    $0x72,%eax
0x0000000000400fdc <+129>: jmpq   0x401098 <phase_3+317>
0x0000000000400fe1 <+134>: mov    $0x6d,%eax
0x0000000000400fe6 <+139>: cmpl   $0x86,0x8(%rsp)
0x0000000000400fee <+147>: je     0x401098 <phase_3+317>
0x0000000000400ff4 <+153>: callq  0x401624 <explode_bomb>
0x0000000000400ff9 <+158>: mov    $0x6d,%eax
0x0000000000400ffe <+163>: jmpq   0x401098 <phase_3+317>
0x0000000000401003 <+168>: mov    $0x75,%eax
0x0000000000401008 <+173>: cmpl   $0x309,0x8(%rsp)
0x0000000000401010 <+181>: je     0x401098 <phase_3+317>
0x0000000000401016 <+187>: callq  0x401624 <explode_bomb>
0x000000000040101b <+192>: mov    $0x75,%eax
0x0000000000401020 <+197>: jmp    0x401098 <phase_3+317>
0x0000000000401022 <+199>: mov    $0x78,%eax
0x0000000000401027 <+204>: cmpl   $0x288,0x8(%rsp)
0x000000000040102f <+212>: je     0x401098 <phase_3+317>
0x0000000000401031 <+214>: callq  0x401624 <explode_bomb>
0x0000000000401036 <+219>: mov    $0x78,%eax
0x000000000040103b <+224>: jmp    0x401098 <phase_3+317>
0x000000000040103d <+226>: mov    $0x7a,%eax
0x0000000000401042 <+231>: cmpl   $0x90,0x8(%rsp)
0x000000000040104a <+239>: je     0x401098 <phase_3+317>
0x000000000040104c <+241>: callq  0x401624 <explode_bomb>
0x0000000000401051 <+246>: mov    $0x7a,%eax
0x0000000000401056 <+251>: jmp    0x401098 <phase_3+317>
0x0000000000401058 <+253>: mov    $0x6b,%eax
0x000000000040105d <+258>: cmpl   $0xe6,0x8(%rsp)
0x0000000000401065 <+266>: je     0x401098 <phase_3+317>
0x0000000000401067 <+268>: callq  0x401624 <explode_bomb>
0x000000000040106c <+273>: mov    $0x6b,%eax
0x0000000000401071 <+278>: jmp    0x401098 <phase_3+317>
0x0000000000401073 <+280>: mov    $0x70,%eax
0x0000000000401078 <+285>: cmpl   $0xf3,0x8(%rsp)
0x0000000000401080 <+293>: je     0x401098 <phase_3+317>
0x0000000000401082 <+295>: callq  0x401624 <explode_bomb>
0x0000000000401087 <+300>: mov    $0x70,%eax
0x000000000040108c <+305>: jmp    0x401098 <phase_3+317>
0x000000000040108e <+307>: callq  0x401624 <explode_bomb>
0x0000000000401093 <+312>: mov    $0x62,%eax
0x0000000000401098 <+317>: cmp    0x7(%rsp),%al
0x000000000040109c <+321>: je     0x4010a3 <phase_3+328>
0x000000000040109e <+323>: callq  0x401624 <explode_bomb>
0x00000000004010a3 <+328>: add    $0x18,%rsp
0x00000000004010a7 <+332>: retq

End of assembler dump.
  
```

Handwritten Annotations:

- Memory Diagram:** Shows 'gcc 2016' and 'libc' memory regions.
- Stack Frame:** Shows 'rsp' (return pointer) and 'rdx' (argument register).
- Function Analysis:**
 - Line <+29>:** `callq 0x400c30 <_isoc99_sscanf@plt>`. Note: `scanf` returns the number of input items successfully read and stores their addresses in the locations pointed to by the arguments. Here, it stores the addresses of `%s`, `%d`, and `%c` in `%si`, `%di`, and `%ci` respectively.
 - Line <+34>:** `cmp $0x2,%eax`. Note: `%eax` is 2. If `%eax < 2`, it goes to `explode_bomb`. If `%eax > 2`, it continues.
 - Line <+44>:** `callq 0x401624 <explode_bomb>`. Note: `(rsp+12) > 7` if `explode_bomb`. If `(rsp+12) < 7`, it continues.
 - Line <+59>:** `jmpq *0x4025d0(,%rax,8)`. Note: `%rax` is 13. `0x4025d0 + 0 = 0x4025d0`. If `%rax` is 13, it goes to `explode_bomb`. If `%rax` is 14, it continues.
 - Line <+71>:** `cmpl $0x22c,0x8(%rsp)`. Note: `0x4025d0 + 0 = 0x4025d0`. If `0x22c` is 14, it goes to `explode_bomb`. If `0x22c` is 15, it continues.
 - Line <+85>:** `callq 0x401624 <explode_bomb>`. Note: `(rsp+6) < 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+100>:** `mov $0x72,%eax`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+105>:** `cmpl $0x1d6,0x8(%rsp)`. Note: `0x1d6` is 470. If `0x1d6` is 470, it goes to `explode_bomb`. If `0x1d6` is 471, it continues.
 - Line <+119>:** `callq 0x401624 <explode_bomb>`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+124>:** `mov $0x72,%eax`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+129>:** `jmpq 0x401098 <phase_3+317>`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+134>:** `mov $0x6d,%eax`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+139>:** `cmpl $0x86,0x8(%rsp)`. Note: `0x86` is 134. If `0x86` is 134, it goes to `explode_bomb`. If `0x86` is 135, it continues.
 - Line <+147>:** `je 0x401098 <phase_3+317>`. Note: `0x86` is 134. If `0x86` is 134, it goes to `explode_bomb`. If `0x86` is 135, it continues.
 - Line <+153>:** `callq 0x401624 <explode_bomb>`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+158>:** `mov $0x6d,%eax`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+163>:** `jmpq 0x401098 <phase_3+317>`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+168>:** `mov $0x75,%eax`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+173>:** `cmpl $0x309,0x8(%rsp)`. Note: `0x309` is 789. If `0x309` is 789, it goes to `explode_bomb`. If `0x309` is 790, it continues.
 - Line <+181>:** `je 0x401098 <phase_3+317>`. Note: `0x309` is 789. If `0x309` is 789, it goes to `explode_bomb`. If `0x309` is 790, it continues.
 - Line <+187>:** `callq 0x401624 <explode_bomb>`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+192>:** `mov $0x75,%eax`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+197>:** `jmp 0x401098 <phase_3+317>`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+199>:** `mov $0x78,%eax`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+204>:** `cmpl $0x288,0x8(%rsp)`. Note: `0x288` is 648. If `0x288` is 648, it goes to `explode_bomb`. If `0x288` is 649, it continues.
 - Line <+212>:** `je 0x401098 <phase_3+317>`. Note: `0x288` is 648. If `0x288` is 648, it goes to `explode_bomb`. If `0x288` is 649, it continues.
 - Line <+214>:** `callq 0x401624 <explode_bomb>`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+219>:** `mov $0x78,%eax`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+224>:** `jmp 0x401098 <phase_3+317>`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+226>:** `mov $0x7a,%eax`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+231>:** `cmpl $0x90,0x8(%rsp)`. Note: `0x90` is 144. If `0x90` is 144, it goes to `explode_bomb`. If `0x90` is 145, it continues.
 - Line <+239>:** `je 0x401098 <phase_3+317>`. Note: `0x90` is 144. If `0x90` is 144, it goes to `explode_bomb`. If `0x90` is 145, it continues.
 - Line <+241>:** `callq 0x401624 <explode_bomb>`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+246>:** `mov $0x7a,%eax`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+251>:** `jmp 0x401098 <phase_3+317>`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+253>:** `mov $0x6b,%eax`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+258>:** `cmpl $0xe6,0x8(%rsp)`. Note: `0xe6` is 230. If `0xe6` is 230, it goes to `explode_bomb`. If `0xe6` is 231, it continues.
 - Line <+266>:** `je 0x401098 <phase_3+317>`. Note: `0xe6` is 230. If `0xe6` is 230, it goes to `explode_bomb`. If `0xe6` is 231, it continues.
 - Line <+268>:** `callq 0x401624 <explode_bomb>`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+273>:** `mov $0x6b,%eax`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+278>:** `jmp 0x401098 <phase_3+317>`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+280>:** `mov $0x70,%eax`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+285>:** `cmpl $0xf3,0x8(%rsp)`. Note: `0xf3` is 243. If `0xf3` is 243, it goes to `explode_bomb`. If `0xf3` is 244, it continues.
 - Line <+293>:** `je 0x401098 <phase_3+317>`. Note: `0xf3` is 243. If `0xf3` is 243, it goes to `explode_bomb`. If `0xf3` is 244, it continues.
 - Line <+295>:** `callq 0x401624 <explode_bomb>`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+300>:** `mov $0x70,%eax`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+305>:** `jmp 0x401098 <phase_3+317>`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+307>:** `callq 0x401624 <explode_bomb>`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+312>:** `mov $0x62,%eax`. Note: `(rsp+6) = 470` if `explode_bomb`. If `(rsp+6) > 470`, it continues.
 - Line <+317>:** `cmp 0x7(%rsp),%al`. Note: `(rsp+1) = 'r'`. If `0x7(%rsp) = 'r'`, it goes to `explode_bomb`. If `0x7(%rsp) > 'r'`, it continues.
 - Line <+321>:** `je 0x4010a3 <phase_3+328>`. Note: `(rsp+1) = 'r'`. If `0x7(%rsp) = 'r'`, it goes to `explode_bomb`. If `0x7(%rsp) > 'r'`, it continues.
 - Line <+323>:** `callq 0x401624 <explode_bomb>`. Note: `(rsp+1) = 'r'`. If `0x7(%rsp) = 'r'`, it goes to `explode_bomb`. If `0x7(%rsp) > 'r'`, it continues.
 - Line <+328>:** `add $0x18,%rsp`. Note: `(rsp+1) = 'r'`. If `0x7(%rsp) = 'r'`, it goes to `explode_bomb`. If `0x7(%rsp) > 'r'`, it continues.
 - Line <+332>:** `retq`. Note: `(rsp+1) = 'r'`. If `0x7(%rsp) = 'r'`, it goes to `explode_bomb`. If `0x7(%rsp) > 'r'`, it continues.

1번째 값: 1
2번째 값: r
3번째 값: 470

<+34> line 을 통해 `eax` 와 2 를 비교하는 것을 알 수 있다. 이때 `eax` 는 <+29> line 의 함수 `scanf` 의 함수값으로서, 이 함수는 입력한 값의 개수가 저장된다. 따라서 <+34>는 입력한 값의 개수가 2 보다 큰지 비교하는 것이다. 이때 2 보다 작거나 같다면 폭탄이 터진다. 따라서 입력값은 반드시 세 개 이상이어야 한다. 이후 `rsp + 0xc` 와 7 을 비교한다. 이때

rsp+0xc 는 %rdx 의 값으로서, 첫 번째 인자를 의미한다. 만약 이것이 7 보다 크다면 폭탄이 터진다. 따라서 첫 번째 인자가 7 보다 작아야 한다는 것을 알 수 있다. 이후 eax 에 rsp+0xc, 즉 첫 번째 인자를 저장한다. 이제 (0x4025d0 + 8*rax)라는 주소가 가리키는 주소로 jump 한다. 본 보고서에서는 rax, 즉 입력할 첫 번째 인자를 1 로 넣고 풀었다. 1 을 넣은 결과값인 0x4025d8 이 가리키는 값을 보면, 0x400fbf 라는 값이 나온다. 해당 주소로 이동하면 된다. 이동을 한 후, eax 에 0x72 를 저장한다. 그 다음 rsp+8, 즉 세 번째 값이 0x1d6, 즉 470 과 다르다면 폭탄이 터진다. 따라서 세 번째 인자가 470 이라는 것을 알 수 있다. 이후 <+317>로 jump 한다. 이제 %al(eax 와 값 동일)과 (rsp+7) 즉 두 번째 인자와 비교한다. 이때 같지 않으면 폭탄이 터진다. 두 번째 인자는 문자 형식이므로, ASCII 코드 상으로 0x72, 즉 114 인 문자를 찾으면, r 인 것을 알 수 있다.

Answer: 1 r 470

<phase_4>

<phase 4>

Dump of assembler code for function phase_4:

```

0x0000000004010e0 <+0>: sub    $0x18,%rsp
0x0000000004010e4 <+4>: lea    0xc(%rsp),%rcx
0x0000000004010e9 <+9>: lea    0x8(%rsp),%rdx
0x0000000004010ee <+14>: mov     $0x40286d,%esi → x/s 0x40286d ⇒ %d %d ∴ 정수 입력을 2번 받은 함수이다.
0x0000000004010f3 <+19>: mov     $0x0,%eax → eax에 0 대입
0x0000000004010f8 <+24>: callq   0x400c30 <__isoc99_sscanf@plt>
0x0000000004010fd <+29>: cmp     $0x2,%eax → eax가 2이면 → eax는 입력받은 인자의 개수를 저장한
0x000000000401100 <+32>: jne     0x40110e <phase_4+46> → eax ≠ 2라면 explode_bomb
0x000000000401102 <+34>: mov     0xc(%rsp),%eax → 2번째 인자 eax에 저장
0x000000000401106 <+38>: sub     $0x2,%eax → eax값이 2개씩
0x000000000401109 <+41>: cmp     $0x2,%eax → 곱한 개로
0x000000000401109 <+41>: jbe     0x401113 <phase_4+51> → eax ≥ 2라면 explode 실행 ∴ eax ≤ 2
0x00000000040110e <+46>: callq   0x401624 <explode_bomb> → 2번째 인자 ≤ 4 이다 ...
0x000000000401113 <+51>: mov     0xc(%rsp),%esi → esi에 2번째 인자 저장
0x000000000401117 <+55>: mov     $0x7,%edi → edi에 7 저장
0x00000000040111c <+60>: callq   0x4010a8 <func4> → func4 실행
0x000000000401121 <+65>: cmp     0x8(%rsp),%eax → 2번째 인자(%rsp+8)과 func4의 결과값 %eax 비교
0x000000000401125 <+69>: je      0x40112c <phase_4+76> → 2번째 인자값 = %eax
0x000000000401127 <+71>: callq   0x401624 <explode_bomb>
0x00000000040112c <+76>: add     $0x18,%rsp
0x000000000401130 <+80>: retq

```

End of assembler dump.

<func4>

Dump of assembler code for function func4:

```

0x0000000004010a8 <+0>: push    %r12
0x0000000004010aa <+2>: push    %rbp
0x0000000004010ab <+3>: push    %rbx
0x0000000004010ac <+4>: mov     %edi,%ebx → edi에 (edi) 저장
0x0000000004010ae <+6>: test    %edi,%edi → edi가 0이면
0x0000000004010b0 <+8>: jle     0x4010d6 <func4+46> → edi ≤ 0이면
0x0000000004010b2 <+10>: mov     %esi,%ebp → esi, %ebp cbp에 2번째 인자 저장
0x0000000004010b4 <+12>: mov     %esi,%eax → eax에도 저장
0x0000000004010b6 <+14>: cmp     $0x1,%edi → edi-1 = 0 이라면
0x0000000004010b9 <+17>: je      0x4010db <func4+51> → edi = 1 이라면
0x0000000004010bb <+19>: lea     -0x1(%rdi),%edi → (rdi-1) 값 edi에 저장
0x0000000004010be <+22>: callq   0x4010a8 <func4> → 재귀호출
0x0000000004010c3 <+27>: lea     0x0(%rbp,%r12,1),%r12 → (%rbp+1), %r12 = (%rbp+1)
0x0000000004010c7 <+31>: lea     -0x2(%rbx),%edi → edi에 (%bx-2) 저장
0x0000000004010ca <+34>: mov     %ebp,%esi → esi에 ebp (2번째 인자) 저장
0x0000000004010cc <+36>: callq   0x4010a8 <func4> → 함수 재호출
0x0000000004010d1 <+41>: add     %r12,%eax → eax에 r12값 더함
0x0000000004010d4 <+44>: jmp     0x4010db <func4+51>
0x0000000004010d6 <+46>: mov     $0x0,%eax → eax에 0 저장 (한 줄)
0x0000000004010db <+51>: pop     %rbx → rbx 반환
0x0000000004010dc <+52>: pop     %rbp
0x0000000004010dd <+53>: pop     %r12
0x0000000004010df <+55>: retq

```

End of assembler dump.

귀찮은 보면

$$func4(k, a) = func4(k-1, a) + func4(k-2, a) + 4$$

이것을 파악할 수 있다.

∴ 계산하면

0, 4, 8, 16, 28, 48, 80, 132

$$\therefore \text{답} = 132, 4$$

func4(%edi, %esi) 실행하기

$$\begin{aligned}
 &\Rightarrow func4(0, 4) = 0 \\
 &func4(1, 4) = 4 \quad (eax=4) \\
 &func4(2, 4) \\
 &\quad rbp = 4 + 4 \\
 &\quad edi = 0 \rightarrow \text{line 43} \text{ line 44} \\
 &\quad esi = 4 \rightarrow func4 = func4(0, 4) \\
 &\quad \text{이 실행된다.}
 \end{aligned}$$

$$\begin{aligned}
 &\therefore \text{line 41} \text{ line 44} \\
 &\quad \%eax = func4(1, 4) = 0 \\
 &\quad \%r12 = 8 \text{ 이다} \\
 &\quad \%eax = \%eax + \%r12 = 8 \\
 &\therefore func4(2, 4) = 8
 \end{aligned}$$

<+29> line 을 보면 eax, 즉 입력값의 개수와 2 를 비교하는 것을 확인할 수 있다. 입력값의 개수가 두 개가 아니라면, 폭탄이 터진다. 다음으로, rsp+0xc, 즉 2 번째 인자를 eax 에 저장한다. 이후 eax 값에 2 를 뺀다. 그리고 그 값을 2 와 비교한다. 이때 eax 가 2 보다 크면 폭탄이 터진다. 즉 2 번째 인자가 4 보다 작거나 같아야 한다는 것을 알 수 있다. 이후 esi 에 2 번째 인자를

저장하고, edi 에 7 을 저장하여 두 레지스터 값을 인자로 갖는 함수 <func4>를 실행한다. 이 함수값이 첫 번째 인자 값과 같아야만, 폭탄이 터지지 않고 통과된다.

Func4 함수를 살펴보면, edi 에 ebx, 즉 7 을 저장하고, ebp 에 2 번째 인자, eax 에도 2 번째 인자를 저장한다. 이때 edi = 0 이라면 바로 함수값을 0 으로 가진 후 함수를 종료하고, edi = 1 이라면 함수값, 즉 eax 를 (두 번째 입력값)으로 저장한 후 함수를 종료한다. 여기서 두 번째 입력값을 4 이하라는 조건에 맞춰, 4 로 기입하여 진행해보도록 하겠다.

함수를 계속 분석하다가, 중간에 func4, 즉 자체 함수가 다시 호출되는 경우가 있다. 즉 이 함수는 재귀 함수임을 파악할 수 있다. 나오는 것을 토대로 점화식을 도출하면,

- 편의상 함수에 인자를 포함시켜, func4(%edi, %esi)라고 하겠다.

| |
|---|
| $\text{Func4}(k,4) = \text{func4}(k-1, 4) + \text{func}(k-2,4) + 4$ |
|---|

가 나온다. Func4(0,4) = 0 이고, func4(1,4) = 4 이므로, 이를 토대로 func(7,4)를 계산하면, 132 가 나온다. 따라서 입력값의 첫 번째 인자는 132 임을 알 수 있다.

Answer: 132, 4

<phase_5>

<phase 5>

```
Dump of assembler code for function phase_5:
0x0000000000401131 <+0>:  push    %rbx
0x0000000000401132 <+1>:  sub     $0x10,%rsp
0x0000000000401136 <+5>:  mov     %rdi,%rbx
0x0000000000401139 <+8>:  callq   0x4013a1 <string_length>
0x000000000040113e <+13>: cmp     $0x6,%eax → EAX=6
0x0000000000401141 <+16>: je      0x401182 <phase_5+81> → eax=6이면 explode_bomb
0x0000000000401143 <+18>: callq   0x401624 <explode_bomb>
0x0000000000401148 <+23>: jmp     0x401182 <phase_5+81>
0x000000000040114a <+25>: movzbl  (%rbx,%rax,1),%edx (rbx+rax) → edx 저장 : <+4> line이벤트, rbx,rbx+1,rbx+2,
0x000000000040114e <+29>: and     $0xf,%edx 1111과 AND연산 ... ,rbx+5까지
0x0000000000401151 <+32>: movzbl  0x402610(%rdx),%edx
0x0000000000401158 <+39>: mov     %dl,(%rsp,%rax,1) → (rax)번째 글자에 대입
0x000000000040115b <+42>: add     $0x1,%rax rax += 1
0x000000000040115f <+46>: cmp     $0x6,%rax
0x0000000000401163 <+50>: jne     0x40114a <phase_5+25>
0x0000000000401165 <+52>: movb    $0x0,0x6(%rsp)
0x000000000040116a <+57>: mov     $0x4025c7,%esi → esi = "devils"
0x000000000040116f <+62>: mov     %rsp,%rdi rsp=rdi
0x0000000000401172 <+65>: callq   0x4013be <strings_not_equal>
0x0000000000401177 <+70>: test    %eax,%eax
0x0000000000401179 <+72>: je      0x401189 <phase_5+88>
0x000000000040117b <+74>: callq   0x401624 <explode_bomb>
0x0000000000401180 <+79>: jmp     0x401189 <phase_5+88>
0x0000000000401182 <+81>: mov     $0x0,%eax → EAX=0
0x0000000000401187 <+86>: jmp     0x40114a <phase_5+25>
0x0000000000401189 <+88>: add     $0x10,%rsp
0x000000000040118d <+92>: pop     %rbx
0x000000000040118e <+93>: retq

End of assembler dump.
```

0x402610에 있는 값은 각인되면
maduiersnfortrbylSo you think you
can stop the bomb with ctrl-c, do you?
devils가 42번, 0x402610에서
즉, <+2> line에서 0xf다 %edx의
AND 값이 2, 5, 12, 4, 15, 7이 나와요.
즉 아스키코드 값이 16x+(2,5,12,4,15,7)이면
1111과 AND 시 2, 5, 12, 4, 15, 7이 나와요.
이 값이 있는 ASCII code를 차인 문자열을 찾아보면
∴ 66, 69, 76, 68, 79, 71
→ b e l d o g.

<+13> 줄에서 `eax` 와 6 을 비교하여, `eax = 6` 이 아닐 경우 폭탄이 터진다. `Eax` 는 문자열의 길이를 계산하는 함수인 `<string_length>`의 함수값을 의미하므로, 입력값이 6 개가 아닐 경우 폭탄이 터지는 구조이다. 그 다음, `rbx+rax` 를 `edx` 에 저장한다. 이후 <+81> line 으로 가서, `eax` 에 0 을 저장한다. 그리고 <+25>로 올라가서, `edx` 에 `(rbx)+rax` 의 값을 저장한다. 이 후 이 값을 0xf, 즉 2 진수로 나타내면 1111 과 AND 연산을 진행한다. 이는 16 으로 나눈 나머지를 의미한다. 이 값을 0x402610 에 더한 값을 `edx` 에 다시 저장하는데, 0x402610 이 가리키는 string 을 찾아보면

"maduiersnfortrbylSo you think you can stop the bomb with ctrl-c, do you?"

라는 문구가 나온다. 이때 0x402610 은 이 string 의 맨처음을 가리키므로, `edx` 는 이 문구에서 (`edx`)번째 문자를 의미한다. 그렇게 입력한 값에 의거하여 6 번을 반복하고, 그 값을 `rsp` 에 저장한다. 그리고 해당 값을 `rdi` 에, 0x4025c7 이 가리키는 문자열을 `esi` 에 저장하고 `<strings_not_equal>` 함수에 넣는다. 이때 0x4025c7 이 가리키는 문자열은 "devils"이다. 즉 devils 와 `rsp` 의 문자열이 같아야만 폭탄이 터지지 않는 것이다. 즉, d, e, v, i, l, s 가 나오는 수가 기입되도록 문자를 넣어주면 된다. 각각 2, 5, 12, 4, 15, 7 번째 숫자들이므로, 16 으로 나눈 나머지가 이 숫자가 되도록 하는 ASCII 코드 상의 문자들을 적어주면 된다. 그러한 문자들을 적어보면, 66, 69, 76, 68, 79, 71 이다. 이를 변환하면, beldog 가 된다.

Answer: beldog

<phase_6>

<phase 6>

Dump of assembler code for function phase_6:

```
0x00000000040118f <+0>: push %r14
0x000000000401191 <+2>: push %r13
0x000000000401193 <+4>: push %r12
0x000000000401195 <+6>: push %rbp
0x000000000401196 <+7>: push %rbx
0x000000000401197 <+8>: sub $0x50,%rsp
0x00000000040119b <+12>: lea 0x30(%rsp),%r13 # r13이 (rsp+48) 주소 값
0x0000000004011a0 <+17>: mov %r13,%rsi
0x0000000004011a3 <+20>: callq 0x40165a <read_six_numbers> # 6개의 숫자를 입력받음...
0x0000000004011a8 <+25>: mov %r13,%r14
0x0000000004011ab <+28>: mov $0x0,%r12d # r12d에 0 대입
0x0000000004011b1 <+34>: mov %r13,%rbp
0x0000000004011b4 <+37>: mov 0x0(%r13),%eax # eax에 0x0(r13) 대입
0x0000000004011b8 <+41>: sub $0x1,%eax # eax-1
0x0000000004011bb <+44>: cmp $0x5,%eax # eax가 5보다 크면 explode... ∴ r13 ≤ 6 ∴ 6개 숫자 모두 6보다 작거나 같아야 함
0x0000000004011be <+47>: jbe 0x4011c5 <phase_6+54>
0x0000000004011c0 <+49>: callq 0x401624 <explode_bomb>
0x0000000004011c5 <+54>: add $0x1,%r12d # r12++
0x0000000004011c9 <+58>: cmp $0x6,%r12d # r12 = 6이면 <+5> ~ <+3> loop 반복
0x0000000004011cd <+62>: je 0x4011f1 <phase_6+98>
0x0000000004011cf <+64>: mov %r12d,%ebx # ebx에 r12 대입 (loop 횟수)
0x0000000004011d2 <+67>: movslq %ebx,%rax # rax = ebx * 4
0x0000000004011d5 <+70>: mov 0x30(%rsp,%rax,4),%eax # eax에 (rsp+4*eax+48) 값
0x0000000004011d9 <+74>: cmp %eax,0x0(%rbp) # 0x0(rbp)과 %eax 비교
0x0000000004011dc <+77>: jne 0x4011e3 <phase_6+84>
0x0000000004011de <+79>: callq 0x401624 <explode_bomb>
0x0000000004011e3 <+84>: add $0x1,%ebx # ebx++
0x0000000004011e6 <+87>: cmp $0x5,%ebx # ebx ≤ 5이면 jump
0x0000000004011e9 <+90>: jle 0x4011d2 <phase_6+67>
0x0000000004011eb <+92>: add $0x4,%r13 # r13에 +4 # 다음 숫자로 넘겨줌
0x0000000004011ef <+96>: jmp 0x4011b1 <phase_6+34>
0x0000000004011f1 <+98>: lea 0x48(%rsp),%rsi # rsi에 (rsp+72) 대입
0x0000000004011f6 <+103>: mov %r14,%rax # rax = r14 * 20, 즉 (r14) * 20
0x0000000004011f9 <+106>: mov $0x7,%ecx
0x0000000004011fe <+111>: mov %ecx,%edx # edx = 7
0x000000000401200 <+113>: sub (%rax),%edx # r14에 대입한 값 * 20을 edx에 저장
0x000000000401202 <+115>: mov %edx,%rax # rax에 저장
0x000000000401204 <+117>: add $0x4,%rax # 4를 더함
0x000000000401208 <+121>: cmp %rsi,%rax # rsi와 rax를 비교, rsi = rsp+48 즉 6개의 문자를 비교
0x00000000040120b <+124>: jne 0x4011fe <phase_6+111>
0x00000000040120d <+126>: mov $0x0,%esi # esi = 0
0x000000000401212 <+131>: jmp 0x401234 <phase_6+165>
0x000000000401214 <+133>: mov 0x8(%rdx),%rdx # 0x10(%rdx) + 0x8 값
0x000000000401218 <+137>: add $0x1,%eax # eax는 1만큼 증가
0x00000000040121b <+140>: cmp %ecx,%eax
0x00000000040121d <+142>: jne 0x401214 <phase_6+133>
0x00000000040121f <+144>: jmp 0x401226 <phase_6+151>
0x000000000401221 <+146>: mov %rdx,%rsi # rdx, 즉 <+133>과 <+144>의 jump
0x000000000401226 <+151>: add $0x4,%rsi # rsi에 +4
0x00000000040122a <+155>: cmp $0x18,%rsi # rsi = 24, 즉 <+133>과 <+144>의 jump
0x000000000401232 <+163>: je 0x401249 <phase_6+186>
0x000000000401234 <+165>: mov 0x30(%rsp,%rsi,1),%ecx # ecx에 (rsp+rsi+0x30) : n번째 byte에 ecx = n번씩
0x000000000401238 <+169>: cmp $0x1,%ecx # ecx ≤ 1이면 jump
0x00000000040123b <+172>: jle 0x401221 <phase_6+146>
0x00000000040123d <+174>: mov $0x1,%eax
0x000000000401242 <+179>: mov $0x6042f0,%edx
0x000000000401247 <+184>: jmp 0x401214 <phase_6+133>
0x000000000401249 <+186>: mov (%rsp),%rbx
0x00000000040124d <+190>: lea 0x8(%rsp),%rax # rax = rsp+8
0x000000000401252 <+195>: lea 0x30(%rsp),%rsi # rsi = rsp+0x30
0x000000000401257 <+200>: mov %rbx,%rcx
0x00000000040125a <+203>: mov (%rax),%rdx
0x00000000040125d <+206>: mov %rdx,0x8(%rcx)
0x000000000401261 <+210>: add $0x8,%rax
0x000000000401265 <+214>: cmp %rsi,%rax
0x000000000401268 <+217>: je 0x40126f <phase_6+224>
0x00000000040126a <+219>: mov %rdx,%rcx
0x00000000040126d <+222>: jmp 0x40125a <phase_6+203>
0x00000000040126f <+224>: movq $0x0,0x8(%rdx) # 0x0(%rdx) = 0
0x000000000401277 <+232>: mov $0x5,%ebp
0x00000000040127c <+237>: mov 0x8(%rbx),%rax
0x000000000401280 <+241>: mov (%rax),%eax
0x000000000401282 <+243>: cmp %eax,%rbx # rbx < eax이면 explode_bomb, ∴ rbx ≥ eax
0x000000000401284 <+245>: jge 0x40128b <phase_6+252>
0x000000000401286 <+247>: callq 0x401624 <explode_bomb>
0x00000000040128b <+252>: mov 0x8(%rbx),%rbx
0x00000000040128f <+256>: sub $0x1,%ebp
0x000000000401292 <+259>: jne 0x40127c <phase_6+237>
0x000000000401294 <+261>: add $0x50,%rsp
0x000000000401298 <+265>: pop %rbx
0x000000000401299 <+266>: pop %rbp
0x00000000040129a <+267>: pop %r12
0x00000000040129c <+269>: pop %r13
0x00000000040129e <+271>: pop %r14
0x0000000004012a0 <+273>: retq
```

node 1 = 686
node 2 = 345
node 3 = 301
node 4 = 110
node 5 = 458
node 6 = 214

→ 2번이 이 값은 현재 배열에서 1번 번째
∴ 3, 6, 5, 2, 4, 1

<read_six_numbers> 함수로 보아, 6 개의 숫자를 입력하는 것을 추측할 수 있다. 맨 처음으로, r13 에 rsp+48 의 주소값을 저장한다. 이후 r12d 에 0, eax 에 0x0(r13)을 저장한다. 이후 eax 에 1 을 뺀 값이 5 보다 크면, 폭탄이 터진다. 따라서 eax, 즉 r13 은 6 보다 작거나 같아야 한다. 이 r13 은 입력한 6 개의 숫자를 하나씩 저장하게 되므로, 입력한 6 개의 숫자 모두 6 보다 작거나

같아야 한다. 이후 r12 에 1 을 더하고, 이 값을 6 과 비교한다. R12=6 이라면 <+98>로 이동한다. 만약 아니라면, r12 를 ebx 에 저장하고, ebx 를 다시 rax 에 저장한다. 이후 $rsp + (4 * rax + 0x30)$ 값을 eax 에 저장한다. 이후 이 값을 $0x0(\%rbp)$ 와 비교하여, 같다면 폭탄이 터진다. 그리고 ebx 0 에 1 을 추가하고 5 와 비교한 후, 다시 <+34>로 돌아가 비교한다. Ebx 가 5 에 도달하면, r13 에 4 를 추가하여 다음 번째의 숫자를 가리키게 한 후 그 값을 다시 rbp 에 넣어 다른 숫자와 비교한다. 즉, 이 구조는 이중 루프로서 입력값이 6 개의 숫자들을 서로 다 비교하여, 같은 숫자가 존재할 경우 폭탄을 터트리는 형식이다. 따라서 6 개 모두 다른 숫자여야 한다.

동일한 지 검사를 하는 루프를 빠져나온 이후, 각각의 값들을 다시 7 에서 뺀 값으로 치환한다. 예를 들어 4 였다면, 그 자리에는 $7-4=3$ 이 들어가게 되는 것이다.

이후, esi 를 0 으로 저장한후 <+165> line 으로 간다. 이때 ecx 에 $rsp+0x30+rsi$ 를 저장한다. 이때 $rsp+0x30$ 은 입력값의 첫 번째 인자의 주소이고, rsi 는 loop 를 돌면서 1 씩 증가한다. 즉, ecx 는 입력값을 의미한다. 입력값이 1 보다 크다면, $0x6042f0$ 에서 $+8*n$ 한 값을 저장한다. 이때 n 은 ecx 의 값을 의미한다. 그리고 그 주소를 $rsp+rsi*2$ 에 저장한다. 이를 총 6 번 반복한다. 이때 $0x6042f0$ 에 있는 값을 확인해보면, <node1>이라고 인덱싱 된 값을 확인할 수 있다. 이를 x/wx 를 통해 확인해보면 "0x000002ae", 즉 686 이다. 이 때 이 주소값은, 1 이 등장하는 번째의 인덱스인 <node k>에서 +8 한 주소에 저장되어있을 것이다.

그렇게 저장된 값들을 <+243>에서 비교하는데, 이때 rbx 와 $rbx+8$ 값인 eax 를 비교한다. 즉, 연속하는 두 숫자의 크기를 비교하는 것이다. 이때 rbx 보다 eax 보다 작으면, 폭탄이 터진다. 즉 rbx 가 eax 보다 크거나 같아야 한다. 이는 전체적으로 해당하므로, 오름차순으로 진행되어야 함을 알 수 있다. 주소값 분석을 통해, node1 = 686, node2 = 545, node3 = 381, node4 = 690, node5 = 458, node6 = 214 이므로, 순서대로 배열하면, 4,1,2,5,3,6 이 된다. 그런데 이 값이 7 에서 뺀 값이 각각 되야 하므로, 이를 적용하면: 3,6,5,2,4,1 이 된다.

Answer: 3, 6, 5, 2, 4, 1

<phase_defused>

<phase_defused>

Dump of assembler code for function phase_defused:

```

0x00000000004017c2 <+0>: sub    $0x68,%rsp
0x00000000004017c6 <+4>: mov    $0x1,%edi
0x00000000004017cb <+9>: callq 0x401560 <send_msg>
0x00000000004017d0 <+14>: cmpl   $0x6,0x202fc5(%rip)
0x00000000004017d7 <+21>: jne    0x401846 <phase_defused+132>
0x00000000004017d9 <+23>: lea    0x10(%rsp),%r8
0x00000000004017de <+28>: lea    0x8(%rsp),%rcx
0x00000000004017e3 <+33>: lea    0xc(%rsp),%rdx
0x00000000004017e8 <+38>: mov    $0x4028b7,%esi → 0x4028b7: "%d %d %s"
0x00000000004017ed <+43>: mov    $0x6048b0,%edi → 0x6048b0: "122 4" → phase-4의 답이다
0x00000000004017f2 <+48>: mov    $0x0,%eax
0x00000000004017f7 <+53>: callq 0x400c30 <_isoc99_sscanf@plt>
0x00000000004017fc <+58>: cmp    $0x3,%eax → eax ≠ 3이면 secret-phase에 못감
0x00000000004017ff <+61>: jne    0x401832 <phase_defused+112>
0x0000000000401801 <+63>: mov    $0x4028c0,%esi
0x0000000000401806 <+68>: lea    0x10(%rsp),%rdi
0x000000000040180b <+73>: callq 0x4013be <strings_not_equal>
0x0000000000401810 <+78>: test   %eax,%eax → strings_not_equal이 0이면 secret-phase에 3점
0x0000000000401812 <+80>: jne    0x401832 <phase_defused+112>
0x0000000000401814 <+82>: mov    $0x402718,%edi
0x0000000000401819 <+87>: callq 0x400b40 <puts@plt>
0x000000000040181e <+92>: mov    $0x402740,%edi
0x0000000000401823 <+97>: callq 0x400b40 <puts@plt>
0x0000000000401828 <+102>: mov    $0x0,%eax
0x000000000040182d <+107>: callq 0x4012df <secret_phase> → secret phase 존재
0x0000000000401832 <+112>: mov    $0x402778,%edi
0x0000000000401837 <+117>: callq 0x400b40 <puts@plt>
0x000000000040183c <+122>: mov    $0x4027a8,%edi
0x0000000000401841 <+127>: callq 0x400b40 <puts@plt>
0x0000000000401846 <+132>: add    $0x68,%rsp
0x000000000040184a <+136>: retq

```

End of assembler dump.

Handwritten notes and annotations:

- phase가 들어올 때마다 증가. (0부터)
- 즉 phase가 6개 동작을 하지 않을 경우 secret phase 도착 불가
- phase-4 답
- 즉 4에 %5, 즉 문자열을 똑같이 입력해야 한다!
- secret phase에 도달하지 못하게 함
- rdi = <+33> line (rsp+0xc) 값
- esi = 0x4028c0에 있는 값 → "DrEvil"
- ∴ phase-4 줄이 DrEvil을 입력하면 secret phase가 열린다.

<secret_phase>

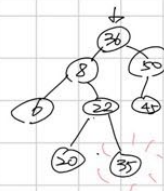
Dump of assembler code for function secret_phase:

```

0x00000000004012df <+0>: push   %rbx
0x00000000004012e0 <+1>: callq 0x40169c <read_line> → 문자열을 입력받는 함수
0x00000000004012e5 <+6>: mov    $0xa,%edx → edx = 10
0x00000000004012ea <+11>: mov    $0x0,%esi → esi = 0
0x00000000004012ef <+16>: mov    %rax,%rdi → rdi = 문자열
0x00000000004012f2 <+19>: callq 0x400c00 <strtol@plt> → 문자면 → 정수로 변환
0x00000000004012f7 <+24>: mov    %rax,%rbx → rbx에 입력값 저장
0x00000000004012fa <+27>: lea    -0x1(%rax),%eax
0x00000000004012fd <+30>: cmp    $0x3e8,%eax → eax > 1000이면 explode bomb ∴ eax ≤ 1000
0x0000000000401302 <+35>: jbe    0x401309 <secret_phase+42>
0x0000000000401304 <+37>: callq 0x401624 <explode_bomb>
0x0000000000401309 <+42>: mov    %ebx,%esi esi = 입력값
0x000000000040130b <+44>: mov    $0x604110,%edi → edi = 0x24 (36) → 0x604110으로부터 +0, +16씩 계산하면 값들이 나오게 되는데, 이는 두 줄씩 4단은 func기서 이용된다
0x0000000000401310 <+49>: callq 0x4012a1 <fun7>
0x0000000000401315 <+54>: cmp    $0x6,%eax fun7 return값 = 6이면 defused 된다.
0x0000000000401318 <+57>: je     0x40131f <secret_phase+64>
0x000000000040131a <+59>: callq 0x401624 <explode_bomb>
0x000000000040131f <+64>: mov    $0x402598,%edi
0x0000000000401324 <+69>: callq 0x400b40 <puts@plt>
0x0000000000401329 <+74>: callq 0x4017c2 <phase_defused>
0x000000000040132e <+79>: pop    %rbx
0x000000000040132f <+80>: retq

```

End of assembler dump.



<fun7>

```

Dump of assembler code for function fun7:
0x0000000004012a1 <+0>:  sub    $0x8,%rsp
0x0000000004012a5 <+4>:  test   %rdi,%rdi  → rdi = 0 이면 jump
0x0000000004012a8 <+7>:  je     0x4012d5 <fun7+52>
0x0000000004012aa <+9>:  mov     (%rdi),%edx  edx = 36
0x0000000004012ac <+11>:  cmp     %esi,%edx  36 ≤ 입력값이면 jump
0x0000000004012ae <+13>:  jle     0x4012bd <fun7+28>
0x0000000004012b0 <+15>:  mov     0x8(%rdi),%rdi  36 > 입력값이면 (rdi+8)로 이동해서 다시 fun7 실행 (재귀)
0x0000000004012b4 <+19>:  callq   0x4012a1 <fun7> 재귀
0x0000000004012b9 <+24>:  add     %eax,%eax  eax = 2*eax
0x0000000004012bb <+26>:  jmp     0x4012da <fun7+57>
0x0000000004012bd <+28>:  mov     $0x0,%eax  eax = 0
0x0000000004012c2 <+33>:  cmp     %esi,%edx  입력값 = 36이면 jump → eax = 0
0x0000000004012c4 <+35>:  je     0x4012da <fun7+57>
0x0000000004012c6 <+37>:  mov     0x10(%rdi),%rdi  36 ≤ 입력값이면 (rdi+16)으로 이동해서 다시 fun7 실행 (재귀)
0x0000000004012ca <+41>:  callq   0x4012a1 <fun7>
0x0000000004012cf <+46>:  lea     0x1(%rax,%rax,1),%eax  eax = 2 * (rax) + 1
0x0000000004012d3 <+50>:  jmp     0x4012da <fun7+57>
0x0000000004012d5 <+52>:  mov     $0xffffffff,%eax
0x0000000004012da <+57>:  add     $0x8,%rsp
0x0000000004012de <+61>:  retq
End of assembler dump.

```

→ 6 이 나중이면 $eax = 2 * eax$
 \downarrow
 $eax = 2 * rax + 1$ ← 순으로 진행되어야 한다.
 \downarrow
 $eax = 2 * rax + 1$ ← 즉, 왼쪽, 오른쪽, 오른쪽을 진행하면
 되고, 이때 나오는 값은 35이다 //

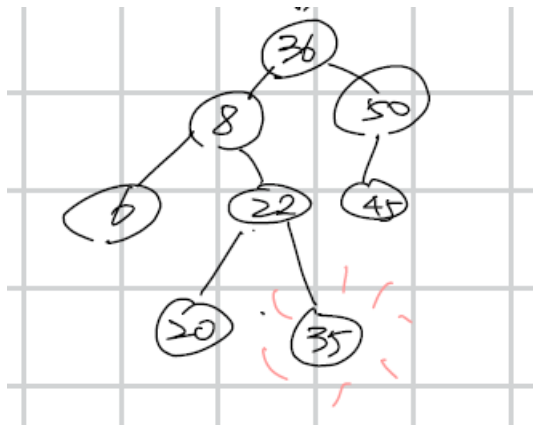
Phase_defused 를 역어셈블하면, <+107> line 에 secret phase 가 존재하는 것을 확인할 수 있다. 하지만 이 secret phase 를 만나기 위해서는 다양한 조건의 jump 들을 피해야 한다. 가장 처음, 6 과 0x60479c 가 가리키는 값이 같지 않으면 secret phase 를 만나지 못한다. phase 들을 점진적으로 풀때마다 0x60479c 가 1 씩 늘어나는 것을 확인할 수 있었고, 이를 통해 0x60479c 가 6, 즉 6 개의 모든 phase 들을 풀지 못하면 secret_phase 를 만나지 못한다는 것을 의미한다.

이후 <+38> line 에서 0x4028b7 을 확인한 결과 "%d %d %s"가 나왔다. 그리고 <+43> line 의 0x6048b0 을 확인한 결과 "132, 4"가 나왔다. 이는 phase_4 의 답으로, phase_4 는 정수 2 개를 입력하는 건데 형식에서 마지막에 %s 가 있는 것을 보아 추가적으로 문자열을 적는다면 secret phase 에 도달 할 수 있을 것이라 생각했다. 그 문자열은 아래에 있었다. <+63>의 0x401832 를 조사해보니, "DrEvil"이라는 문자가 나왔다. 즉 phase_4 에서 "132 4" 뒤에 DrEvil 을 추가로 적으면 secret phase 에 들어갈 수 있는 것이다.

Secret phase 를 역어셈블하면, read_line 과 strtol 이라는 함수가 나온다. 각각 문자열을 입력받는 함수와 그 문자열을 정수로 변환하는 함수이다. 코드를 보면, strtol 의 함수값인 rax 에 1 을 뺀 값이 1000 보다 크면 폭탄이 터지는 것을 확인할 수 있었다. 즉, 입력값이 **1001** 보다 작아야 한다. 이후 입력값인 ebx 를 esi 에, 0x604110 이 가리키는 값이 0x24 를 edi 에 저장한 후 fun7 함수를 작동한다. 이때 fun7 함수의 값이 6 이어야만 defuse 가 되는 것을 확인할 수 있다.

Fun7 함수를 보면, 입력값인 36 을 기준으로 또다른 입력값이 36 보다 작으면 본 위치로부터 +8 주소로, 36 보다 크거나 같으면 본 위치로부터 +16 주소로 가서 다시 fun7 을 재귀한다. 즉 크기에 따라 분기하는 특징을 통해 이 함수가 이진 탐색 트리의 구조를 띄고 있음을 확인할 수

있다. 그리고 각각 `eax` 를 2 배, 2 배+1 한다. 이진 탐색 트리 구조상으로 생각을 해본다면, 왼쪽으로 가면 2 배, 오른쪽으로 가면 2 배 +1 인 것이다. 우리는 함숫값인 `eax` 가 중요하므로, `eax` 가 6 이 나오려면 순서대로 2 배+1, 2 배+1, 2 배가 되어야 한다. 이때 이 함수는 재귀이므로 역순으로 진행하면, 왼쪽, 오른쪽, 오른쪽으로 가면 나오는 값을 넣으면 된다는 것을 알 수 있다. `0x604110` 으로부터 +8, +16 씩 추가하면서 해당 값을 확인해 본 결과, 35 임을 알 수 있었다. 이때 원소가 일부 생략된 대략적인 트리의 구조는 다음과 같다.



Answer: 35